INFERENCE NEURAL NETWORK HARDWARE ACCELERATION TECHNIQUES

BY

YU-HSUAN TSENG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Associate Professor Nam Sung Kim

Abstract

A number of recent researches focus on designing accelerators for popular deep learning algorithms. Most of these algorithms heavily involve matrix multiplication. As a result, building a neural processing unit (NPU) beside the CPU to accelerate matrix multiplication is a popular approach. The NPU helps reduce the work done by the CPU, and often operates in parallel with the CPU, so in general, introducing the NPU gains performance. Furthermore, the NPU itself can be accelerated due to the fact that the majority operation in the NPU is multiply-add. As a result, in this project, we propose two methods to accelerate the NPU: (1) Replace the digital multiply-add unit in the NPU with time-domain analog and digital mixed-signal multiply-add unit. (2) Replace the multiply-add operation with a CRC hash table lookup. The results show that the first proposed method is not as competitive because of the long delay and high energy consumption of the unit. The second method is more promising in that it improves the energy by 1.96× with accuracy drop within 1.2%.

To my ever-supportive family.

Acknowledgments

This project would not have been possible without the support of many people. Many thanks to my adviser, Nam Sung Kim, for guiding me and giving key insights when I was struggling on the projects, and also giving me very useful advice on the job hunt which led me to my dream job. Thanks to Zhenghong Liu for providing numerous kinds of help through the projects. And finally, thanks to my parents and numerous friends for their support and love.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DL	Deep Learning.
DNN	Deep Neural Network.
NN	Neural Network.
CNN	Convolutional Neural Network.
NPU	Neural Processing Unit.
MSMA	Mixed-Signal Multiply-Add Unit.
DMA	Digital Multiply-Add Unit.
DTC	Digital-to-Time Converter.
TDC	Time-to-Digital Converter.
MSB	Most Significant Bit.
LSB	Least Significant Bit.
CRC	Cyclic Redundancy Check.
ALU	Arithmetic Logic Unit.
FPU	Floating Point Unit.

Chapter 1

Introduction

In recent years, performance enhancement of machines or applications by increasing CPU clock frequency has reached a point of saturation due to the unresolved issues of thermal power leakage and the fact that the growth of memory clock frequency cannot keep up with the CPU clock frequency. Recent research tends to explore architectural improvements to get better performance with lower power. Several applications with high computational complexity and memory requirements have been mapped to modified parallel architectures to deliver high performance and throughput [Esmaeilzadeh et al., 2011].

One very popular area that needs such architectural improvement is deep learning (DL). Deep neural networks (DNNs) have led to breakthroughs in recent years, such as significantly reducing error rate in ImageNet Large Scale Visual Recognition Challenge [Krizhevsky et al., 2012], beating traditional speech recognition methods [Hinton et al., 2012], and defeating human in Go.

These achievements would not be made possible without the improvements in the computer architecture area. The concept of DNN has been around since the 1960s [Minsky and Papert, 1969]; however, due to lack of computation power, the performance did not live up to the hype until around 2010, when the computing hardware was mature enough to support the high demand on computation and memory. The phenomenon is illustrated in a virtuous cycle in Figure 1.1. When the datasets get larger, various algorithms for training and using the data are then invented, which call on demand for more powerful architectures. As the architectures improve, more and more datasets can then be fed, and then more advanced DNN models and algorithms pop up. These three key factors keep pushing one another and form a self-reinforcing cycle. Therefore, as DL has gained popularity recently, more and more architectural researches focus on improving the architecture to fulfill the large - and still increasing - computation and memory demand [Reagen et al., 2017].

1.1 Architectural Platforms for Deep Learning

Among all the architectural enhancements that have been tried, one is through the use of a heterogeneous computation platform (Table 1.1). For example, instructions for the critical and computation intensive parts of an algorithm can be offloaded for execution to another parallelized architecture (for example, FPGAs or GPUs) to achieve more significant speed-ups as compared with a simple CPU implementation. When

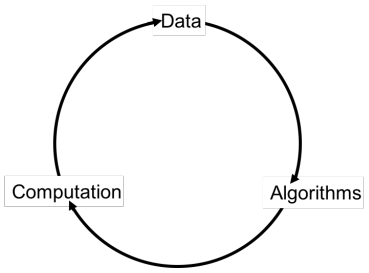


Figure 1.1: The virtuous cycle of DNN.

Table 1.1: List of platforms and their pros & cons for DL applications.

Platform	General Purpose Hardware (CPU)	General Purpose Hardware (GPU)	Dedicated Hardware (ASIC)
Advantages	Flexible: fast development Powerful	Flexible: fast development Powerful High parallelism	Efficient: low power Compatible
Drawbacks	Power hungry Slow	Power hungry	Risk of obsolete design

applying heterogeneous platforms to DL applications, two factors need to be considered: flexibility and efficiency.

Flexibility refers to the programmability of the hardware, which can be performed well on CPUs and GPUs. These hardwares are good for developing and can avoid the high non-recurring engineering (NRE) cost of designing a new dedicated hardware. This feature is especially important for DL because of the rapid change in algorithms. GPUs, in particular, are suitable for DL since neural networks (NNs) are inherently highly parallel. In addition, most of the NN propagations are basically huge matrix multiplications and therefore have high regularity which GPUs are perfect fits.

On the other hand, a dedicated hardware, or an application specific hardware, can achieve high efficiency because the hardware can be optimized to be highly compatible with the targeted applications or the algorithms. Recently, dedicated hardwares in DL have gained a lot of attention because of the following strengths:

- **Low instruction cost**: Like the VLSI design, a specific purpose hardware can have very few instructions with lots of operations inside one single instruction. The overhead of fetch and decode is then significantly reduced. In addition, fewer logics are needed to solve control and data hazards but the architecture can still achieve high parallelism at the same time. For example, in the Tensor Processing Unit Google designed [Jouppi et al., 2017], there are only about a dozen instructions, and one of them is *MatrixMultiply*, which could be equivalent to thousands of instructions if the matrix multiplication would have been done on a CPU.

- **Optimized data type**: If the data type needed in the dedicated hardware is simple, then the com-

putation logic can be made much simpler too. As a consequence the area and power are significantly reduced. The inference part of NNs takes huge advantage of that because many of the applications do not require that much precision in the data in order to have a decent accuracy. Research has shown that 8-bit integer multiplication is enough for most of the NNs these days [Jouppi et al., 2017]. For this major reason, our project (described in Chapter 2) is made possible by implementing 8-bit multiply-add operation.

- **Tightly coupled memory**: An on-chip memory can be explicitly made in a dedicated hardware, and thus reduce the latency and the power of memory fetch. Our project in Chapter 3 takes great advantage of this.

Furthermore, due to the following properties of DL, it is found to benefit from dedicated hardware substantially:

- **Execution time dominated by few operations**: The most popular and widely used DL techniques are convolutional NNs (CNNs) and fully connected NNs [Jouppi et al., 2017]. In these techniques, the dominant operations are convolution and matrix-multiplication, respectively [Reagen et al., 2017, p. 37]. Moreover, the convolution operation can be transformed into matrix-multiplication. Therefore, matrix-multiplication plays a major part in DL and is worth optimizing in a dedicated hardware.

- **Determined control flow and memory accesses**: In CNNs and fully-connected NNs, the loop counts, control flow and memory accesses are already known before the execution. This determined and regular property allows the dedicated hardware to be optimized without speculation; moreover, the memory access latency can be hidden, and so optimal parallelization can be achieved.

1.2 Deep Learning and Approximate Computing

To further gain performance and energy efficiency, one technique, called approximate computing, is often used. In general, approximate computing can achieve substantial energy savings [Sampson et al., 2011]. Many modern applications, such as speech recognition, augmented reality, and data mining, can tolerate inexact computing in critical parts of computation. In fact, the applications that can apply DL can all be approximated because DL is essentially one way of approximation. For example, in the paper [Esmaeilzadeh et al., 2012], a dedicated DNN hardware is used to approximate approximable code regions in various applications with a quality loss of at most 9.6%. Conversely, approximate computing techniques can be used in DL since it can tolerate errors and noise. This indicates that we can "accelerate" a "DNN accelerator" by

applying approximate computing in the accelerator. In this thesis, we explore two approximate computing techniques to accelerate the multiply-add operation, which is the base operation of matrix-multiplication and thus the dominant operation in DNNs.

1.3 Training and Inference in Deep Learning

In DNNs, there are two major phases: the feed-forward phase and the back-propagation phase. In the feed-forward phase, the outputs are given by calculation from the given inputs and the fixed model parameters. The back-propagation phase adjusts the parameters based on the partial derivatives of the loss function with respect to the parameters. The training of DL involves both phases, while the inference only requires the feed-forward phase. Therefore, when designing hardware to work on DL, the methodology is different for training versus inference. Inference is inherently more suitable for a small accelerator due to the following properties:

- **Fixed model parameters**: Fixed parameters means no write operation is needed; thus, dedicated hardwares can be simplified. Moreover, in our proposed technique described in Chapter 3 which replaces multiply-add operation with hash table lookup, fixed model parameters give much higher hit rate since the hash key is calculated from the input neuron as well as the parameters. If the parameters are fixed, then the chance of getting the same hash key (resulting in a hit) is much higher.
- **Low precision requirement**: Unlike training, feed-forward process does not require high precision to calculate the partial derivatives. As a result, the data can be truncated to fewer bits or the data type can be quantized from floating point to integer. Fewer bits can often reduce the operation time quadratically, as shown later in Chapter 2. Going from floating point to integer reduces the circuit complexity massively (also shown in Chapter 2).

Based on the above properties, we propose two acceleration techniques for DNN accelerators. The techniques will be briefly discussed in the next section.

1.4 Project Goal

In this thesis, we explore the possibility of optimizing the DNN accelerators by replacing the multiply-add operations with more efficient designs in exchange of little accuracy lost. The first method (Chapter 2) moves the operation from digital-domain to time-domain. In time-domain, only single-bit signal is needed and the multiply-add operations are turned into simple configurations and concatenations. Furthermore, the logic of

capping the data at the maximum value when overflow occurs is also simpler than in digital-domain. All of the above mentioned benefits indicate the potential for improvement. Therefore, a mixed-signal multiply-add unit is being designed, where the digital input data are mapped to time-domain, and then the result value is decoded back to digital-domain.

The second method (Chapter 3) directly eliminates the operation by substituting it with table lookup. Small, tightly coupled memory of the accelerator gives super-fast memory access as well as low energy. In our simulation, floating point multiply-add operation takes about 20 pJ, and a 16 kB hash table access takes about 2.8 pJ, which is about one seventh the energy. In addition, model parameters distribution analysis shows that when applying DNN applications, there is potentially high hit rate with small hash table. These analysis results lead us to incorporating hash table into the accelerator in the hope of completely eliminating energy-consuming multiply-add operations.

Chapter 2

Time-Domain Analog and Digital Mixed-Signal Multiply-Add

2.1 Overview

Time-domain analog and digital mixed-signal processing [Miyashita et al., 2014] is a hybrid method of analog and digital signal processing that aims to obtain the advantages from both sides. Using single time-delay analog signal to represent the value gives potentially lower power over digital circuit, which requires lots of bits to represent a single value. On the other hand, digital circuit has good performance, speed, and accuracy. By operating the power-hungry computation part in the time-domain and the data transfer part in the digital-domain, we attempt to enjoy benefits from both worlds and have the best balance of speed-power trade-off.

2.2 Introduction

Figure 2.1 shows the time-domain analog and digital mixed-signal design abstract. The data in digital-domain are first transformed to time-domain; then the computation is done in time-domain and finally the data are transformed back to digital-domain. In time-domain, the data value is represented by the delay of the rising edge of the data signal from the clock signal, namely, the time difference between the clock edge and the data transition edge. The digital-to-time converter (DTC) converts the digital data value into a signal with corresponding rising edge delay. The time-domain computation has a huge advantage because the operation is very simple to implement. For instance, the addition in time-domain is concatenation, and maximum/minimum value can be easily obtained from an and/or gate. In our project, we implement a time-domain analog and digital mixed-signal multiply-add unit (MSMA), which is simpler (and therefore has smaller area and power) than a circuit of a single multiply unit and an add unit.

One major downside of manipulating time-domain which is not negligible is the inexact computation

Figure 2.1: Digital to time to digital mixed-signal design abstract. Reproduced from [Esmaeilzadeh et al., 2012].

due to the delay of the components in the circuit. All the components in the circuit, including transistors, resistors, capacitors, and even wires, contribute to the rising delay which are then considered noise. To make matters worse, no same components contribute to the same delays, which makes the noise unpredictable. To get around with this, the base delay (T_{db}), which represents the value one in time-domain, has to be large enough so that the effect of noise can be ignored, or the application being applied to has to be able to accept inaccuracy. The former solution is considered not so effective since large delay means low performance.

Our MSMA operates data in 8-bit by 8-bit integer format. There are two reasons for choosing this format:

- **Hard to support floating point format in time-domain**: Floating point multiplication and addition requires computations in exponential scale, which is not very tolerant to inaccuracy. For example, when two numbers with different exponents are to be added, first they have to be aligned. As mentioned before, the computation in time-domain is not as accurate. If the alignment is off by one value, the result of addition will be completely different. The same reason applies to multiplication as well. As a result, we choose integer to be our format.

- **Long data length results in long delay**: The simulation shows that to manipulate on 8-bit data, a delay of 250 ns is needed. As the data size grows, the delay grows quadratically. For 16-bit data, 62213 ns of delay is needed, which is approximately 250^2, and hence 16 and longer bits operation is not considered to be implemented.

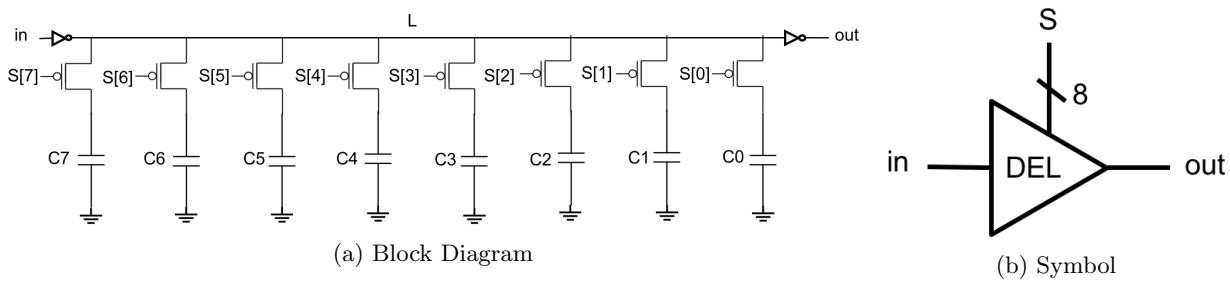
- **8-bit integer is enough for inference NN**: As discussed in Section 1.1 and Section 1.3, 8-bit integer data type is good enough for inference NNs.

Since 8-bit integer format is different from common format used in NNs, quantization is needed to transform the data from floating points to 8-bit integers. Hence, part of the information carried in the data is already lost at this stage.

The rest of the chapter is ordered as follows: Section 2.3 describes the circuit design in detail. Section 2.4 lists the results of simulation and compares them with the results of the digital multiply-add unit (DMA). Section 2.5 summarizes the strengths and weaknesses of the MSMA and suggests possible future modification.

2.3 Implementation

The goal of the MSMA is to lower the energy of matrix-multiplication being performed in the NPU. We first design a digital-to-time converter (DTC) and time-to-digital converter (TDC) pair to transform the



(a) Block Diagram

(b) Symbol

Figure 2.2: Delay unit.

Table 2.1: Configuration of delay unit.

Bit	7	6	5	4	3	2	1	0
PMOS Width (μm)	24	12	6	3	1.9	1.4	1	0.6
Capacitance (pF)	12.8	6.4	3.2	1.6	0.8	0.4	0.2	0.1

data between digital-domain and time-domain. In order to achieve this, a delay unit is needed. Then, we concatenate two DTCs to form an adder, as well as configure the delay value in one of the DTCs to form a multiplier. These two DTCs together make the circuit a MSMA. In addition, we introduce a saturation detect unit to detect overflow and saturate the value upon the occurrence of overflow.

2.3.1 Delay Unit

In order to have a DTC unit which converts the data to corresponding rising edge delay, first we have to design a configurable delay unit. We design such circuit using eight parallel capacitors with value $128C, 64C, 32C, 16C, 8C, 4C, 2C, C$, where C is the base capacitance. The capacitors are turned on and off by the PMOS switch, and therefore the effective capacitance is the sum of the turned on capacitor value. As a result, we can think of the delay unit as configured by an 8-bit data signal S, as shown in Figure 2.2. Ideally, since the delay is proportional to capacitance we can configure the delay from 0, T_{db} to $255T_{db}$, where T_{db} is the base delay time. However, in reality the delay does not go linear with the value of effective capacitance because the PMOS has resistance which affects the delay time. To resolve this problem, we tune the width of the PMOS so that the delay is by and large proportional to the capacitance. The width of the PMOS and the capacitance value are shown in Table 2.1 and the delay to capacitance relationship is shown in Figure 2.3. In this project, the base delay T_{db} is about 0.38 ns.

2.3.2 Digital-to-Time Converter

Next we build a digital-to-time converter based on the delay unit. The circuit diagram of a 1-bit DTC is shown in Figure 2.4a. DTC converts the data by deciding whether to delay the incoming clk signal based on

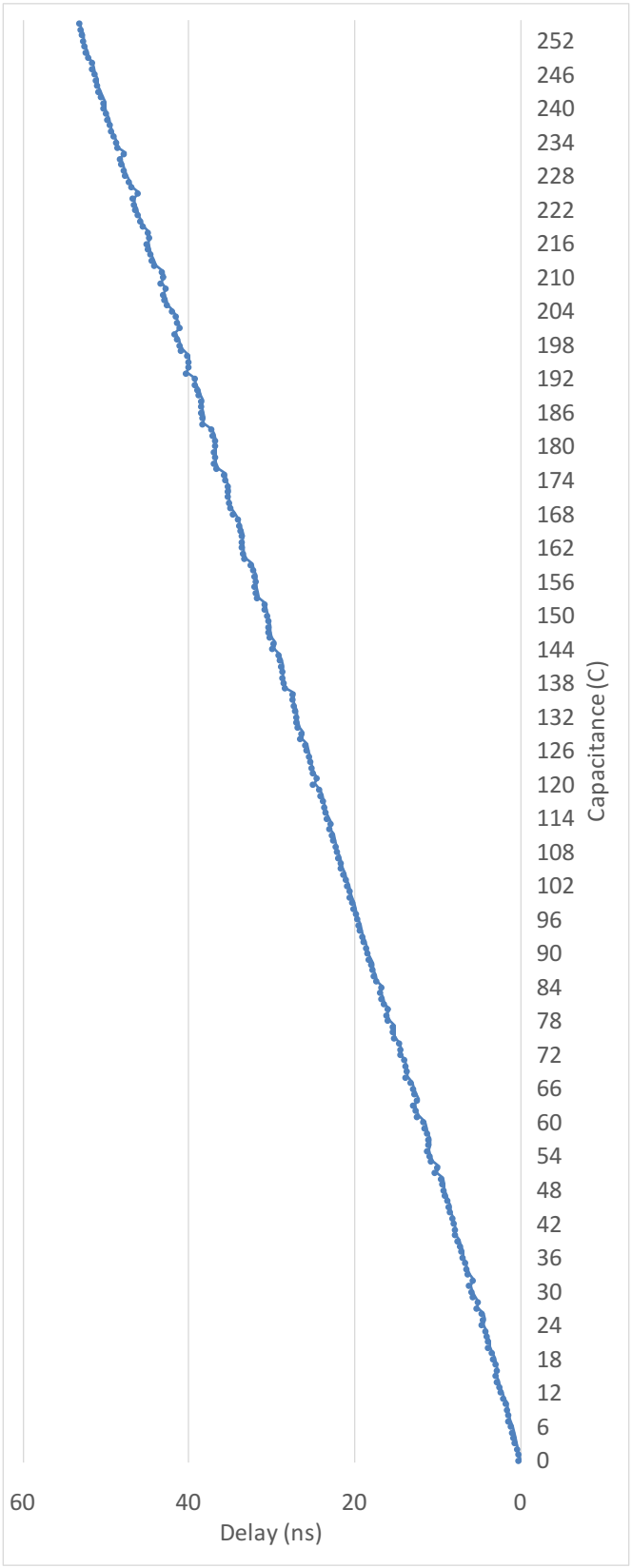
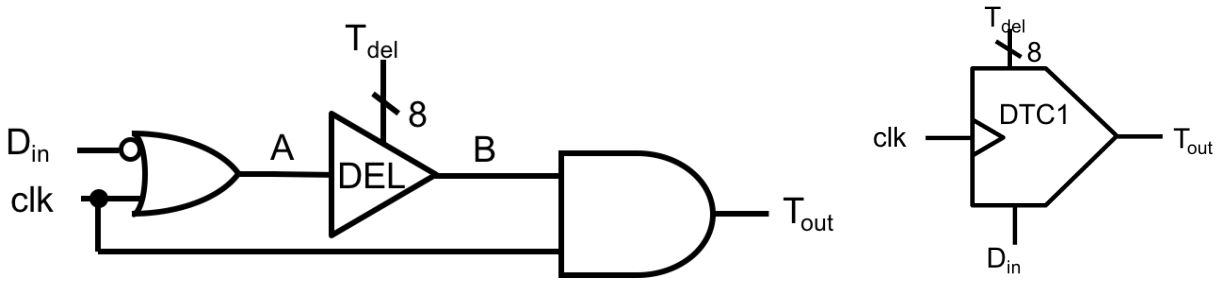


Figure 2.3: Delay to capacitance.



(a) Block Diagram

(b) Symbol

Figure 2.4: 1-bit digital-to-time converter

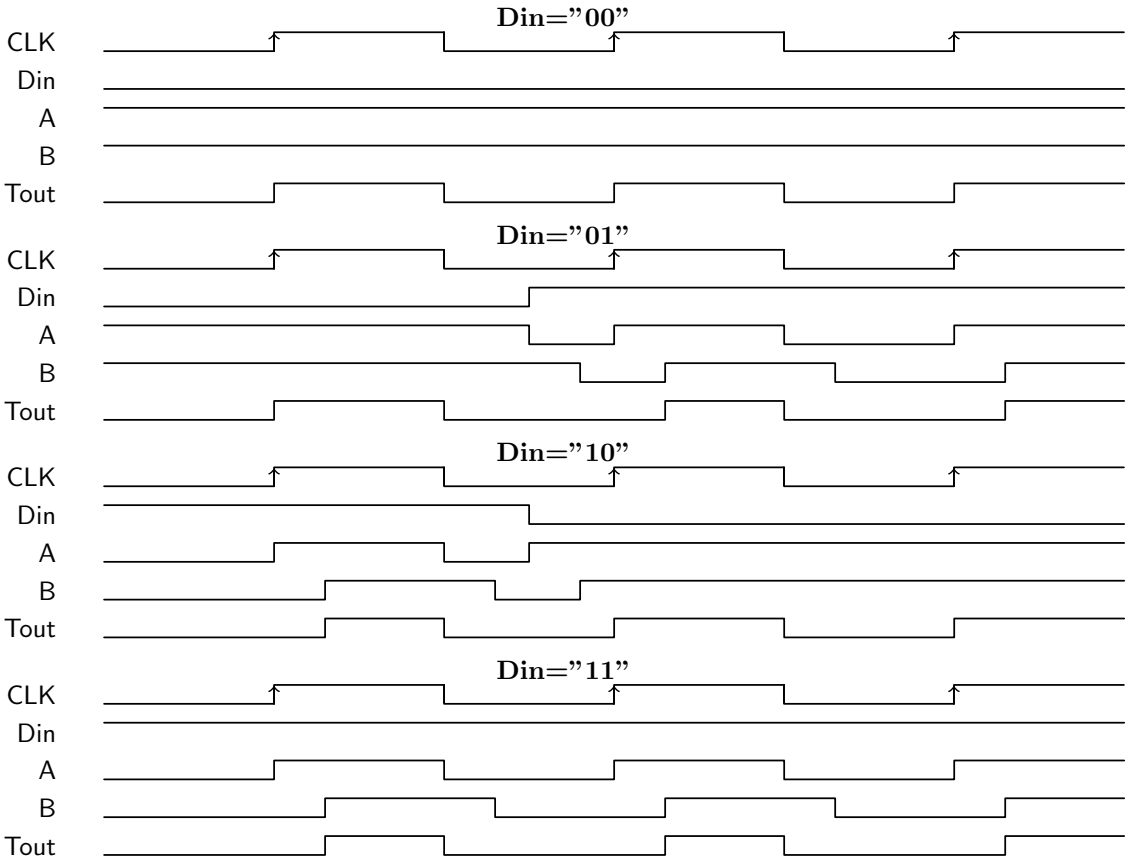
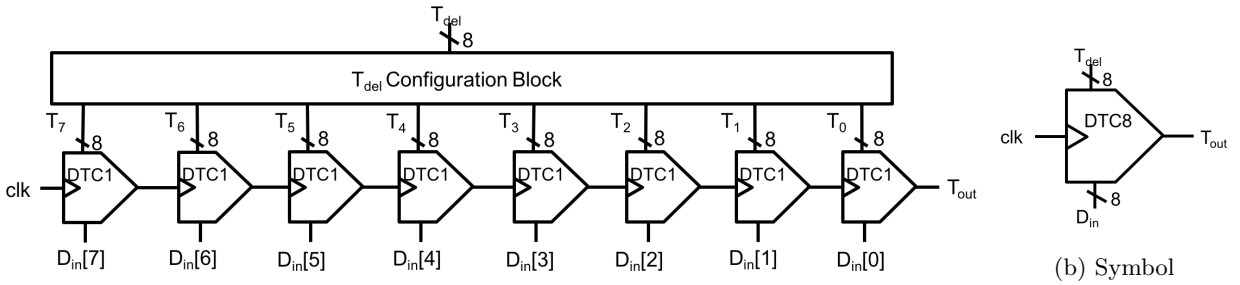


Figure 2.5: DTC timing diagram.



(a) Block Diagram

(b) Symbol

Figure 2.6: 8-bit digital-to-time converter.

the value of D_{in}. If D_{in} equals zero, then the output T_{out} is the same as the incoming signal clk. Otherwise if D_{in} equals one, then T_{out} will be delayed by $T_{del} \times T_{db}$, where T_{del} is configured outside DTC.

The corresponding timing diagram is shown in Figure 2.5. $D_{in} = x_1x_2$ means at the first clock edge D_{in} is x_1, and at the second clock edge D_{in} is x_2. From $D_{in} = 01$ and $D_{in} = 10$, we can see that there is a setup time requirement for D_{in}: D_{in} has to be ready for at least $T_{del} \times T_{db}$ before clk rises. That is, $T_{setup} = T_{del} \times T_{db}$.

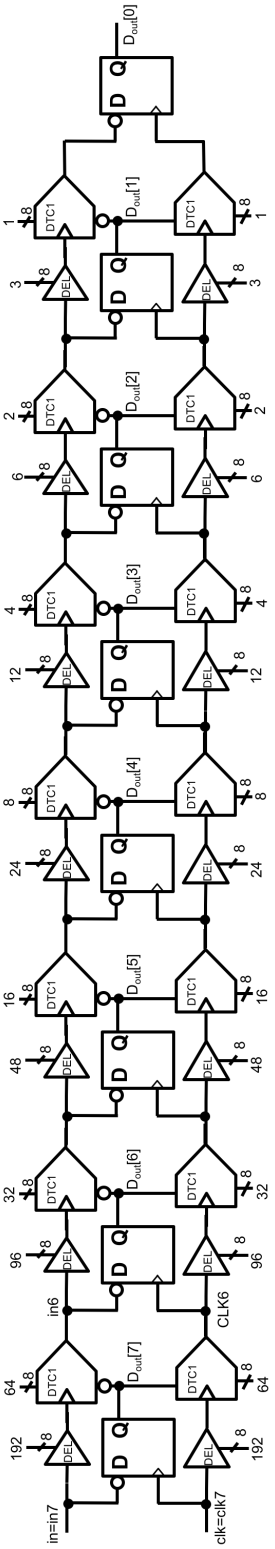
An 8-bit DTC is done by cascading 8 identical 1-bit DTCs where each DTC represents one bit of the 8-bit data, from most significant bit (MSB) to least significant bit (LSB). The block diagram of an 8-bit DTC is in Figure 2.6a. In order for each DTC to represent its own bit position, the delay configuration for each DTC, T_i, must be $2^i \times T_{del}$. The calculation can be done by simple left-shift operations. However, overflow can occur in left-shift, so we have to cap T_i to the maximum value 255 when overflow happens. In order to achieve this, all the bits in T_i are ORed with the bit that will be lost after shifting. The expression of T_i is in Equation (2.1) ("|" is the bitwise or operation).

$$T_i = \begin{cases} T_{del}, & \text{if } i = 0 \\ (8\{T_{i-1}[7]\})|(T_{i-1} << 1), & \text{otherwise} \end{cases} \tag{2.1}$$

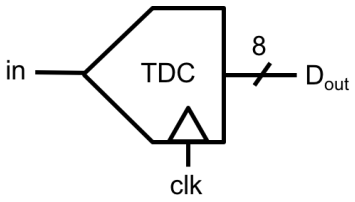
2.3.3 Time-to-Digital Converter

We implement the time-to-digital converter with binary-search algorithm. The search starts from the (MSB), detects the value, and then moves on to the second significant bit to detect the value, until finally all the bit values are detected. The block diagram of an 8-bit TDC is in Figure 2.7.

In order to detect the MSB, first the incoming clk signal should be set in the middle of the data value range (more on this in section 2.3.4). The detection is done by a flip-flop, which captures the bit value of the data signal in when the clock signal clk triggers. We can think of the detection as comparing the input



(a) Block Diagram



(b) Symbol

Figure 2.7: Time-to-digital converter.

rising time T_{in} with the clock rising time T_{clk}.

Next, the detection of the second significant bit is done by comparing the value of the two equations:

$$T_{in} - D_{out}[7] \times 2^7 \times T_{db}$$

$$T_{clk} - 2^6 \times T_{db}$$

Since time cannot be subtracted, we add $D_{out}[7] \times 2^6 \times T_{db} + 2^6 \times T_{db}$ to both equations. The equations then become:

$$T_{in} - D_{out}[7] \times 2^7 \times T_{db} + D_{out}[7] \times 2^6 \times T_{db} + 2^6 \times T_{db} \;=\; T_{in} + \overline{D_{out}[7]} \times 2^6 \times T_{db} \tag{2.2}$$

$$T_{clk} - 2^6 \times T_{db} + D_{out}[7] \times 2^6 \times T_{db} + 2^6 \times T_{db} \;=\; T_{clk} + D_{out}[7] \times 2^6 \times T_{db} \tag{2.3}$$

The resulting equations are almost symmetric except that $D_{out}[7]$ is inverted in Equation 2.2. As a consequence, we use two DTCs, one for *in* (DTC_{in}) and one for *clk* (DTC_{clk}), to "add time" to the signal.

In addition, because of the setup time requirement in DTC (mentioned in section 2.3.2), we introduce a delay unit between *in* and the DTC. There are two requirements for the delay (T_{wait6}) of the delay unit:

- The clock signal for DTC_{in} must trigger after the data signal is ready. The data signal is ready after *clk* triggers the flip-flop. Therefore, *in* must be delayed by the max difference between *in* and *clk*, which is $\frac{2^8-1}{2} \times T_{db}$.
- The data signal should be ready for T_{setup} before the clock signal triggers. Therefore, *in* should be furthermore delayed by $T_{setup} = 2^6 \times T_{db}$.

Adding up the two requirements, we get:

$$T_{wait6} \geq \frac{2^8 - 1}{2} \times T_{db} + 2^6 \times T_{db}$$

We pick $T_{wait6} = (2^7 + 2^6) \times T_{db}$ because in the circuit this can be done by simple *shift* and *or* operations. Furthermore, we add the same delay unit between *clk* and DTC_{clk} in order to add the same delay for time comparison.

For the third significant bit we cascade the same circuit except that the T_{del} for the DTC now becomes 2^5 and the delay T_{wait5} becomes $(2^6 + 2^5) \times T_{db}$. We keep cascading the circuits until the LSB is found.

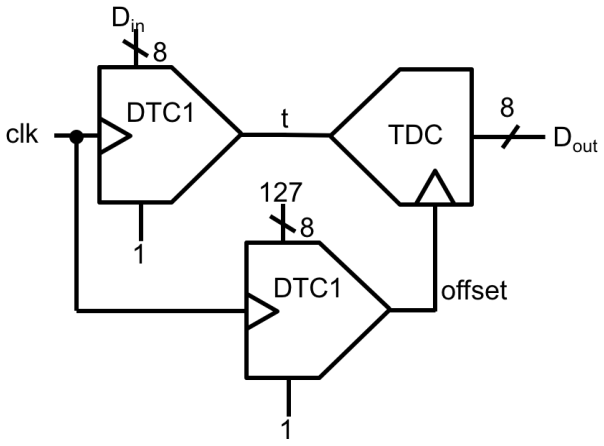
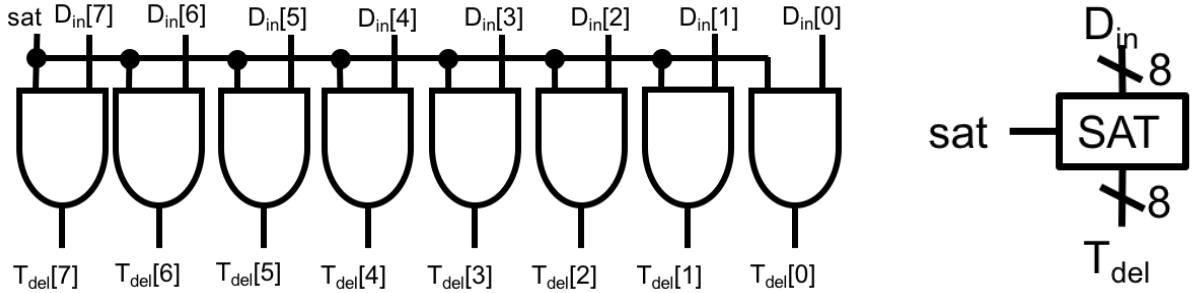


Figure 2.8: DTC-TDC.



(a) Block Diagram

(b) Symbol

Figure 2.9: Saturation unit.

2.3.4 DTC-TDC

The realization of the DTC-TDC unit is illustrated in Figure 2.8. This unit does not do any computation; it is just an encode-decode pair. Therefore, this circuit is used only for simulation and debugging.

In this circuit, D_{in} is transformed to time-domain signal t and then transformed back to digital-domain D_{out}. As mentioned in section 2.3.3, the TDC unit uses binary search from MSB to LSB to decode. Therefore, the clock feeding into the TDC needs to be offset to the middle value of the range to detect MSB. The middle value is $\frac{255}{2} = 127.5$. Since our project targets approximable NN applications and due to the noise caused in the circuit, there is no much difference between 127.5 and 127. As a result, we use a DTC to delay *clk* by $127 \times T_{db}$.

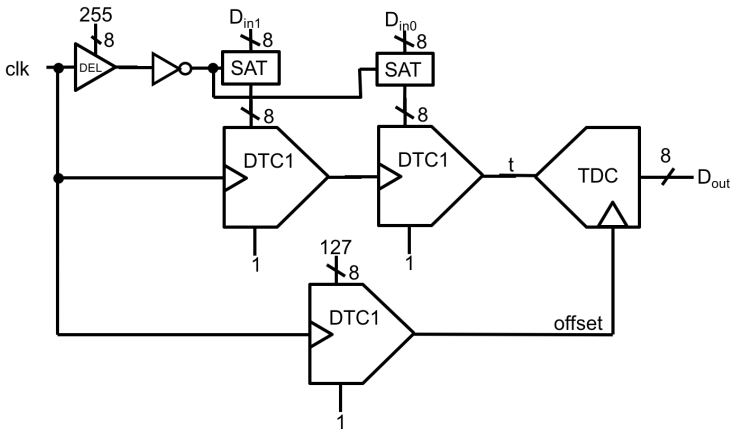


Figure 2.10: Adder.

2.3.5 Saturation Logic

Since overflow occurs in both addition and multiplication, we design a saturation circuit (Figure 2.9) that turns off all the transistor switches once the delay time is $255 \times T_{db}$. This way, the rising edge will be forced to immediately propagate to the TDC if the rising edge delay time exceeds $255 \times T_{db}$ and therefore the data value is capped at 255 in time-domain. The signal *sat* comes from the *clk* delayed by $255 \times T_{db}$.

2.3.6 Adder

Provided circuits from previous sections, we combine them together to build an adder. The block diagram is shown in Figure 2.10. Given that the addition in time-domain is cascading the time signal, two 1-bit DTCs are cascaded. In addition, as we discussed in section 2.3.4, an additional DTC is needed for clock offset. Furthermore, to address the overflow issue, we add two saturation circuits to shut down the two DTCs when overflow occurs. In this way, the decoded D_{out} will be the saturation value 255.

2.3.7 Multiplier

The multiplier (Figure 2.11) is similar to the DTC-TDC unit except that 1-bit DTC is replaced with an 8-bit DTC to realize multiplication. In addition, since overflow occurs in multiplication, a saturation logic is added to saturate the output of the circuit.

2.3.8 MSMA (Multiply-Add Unit)

Similar to the modification done to the DTC-TDC unit to form a multiplier, the MSMA (Figure 2.12) is basically a modification of the adder. One of the 1-bit DTCs is changed to an 8-bit DTC to provide the

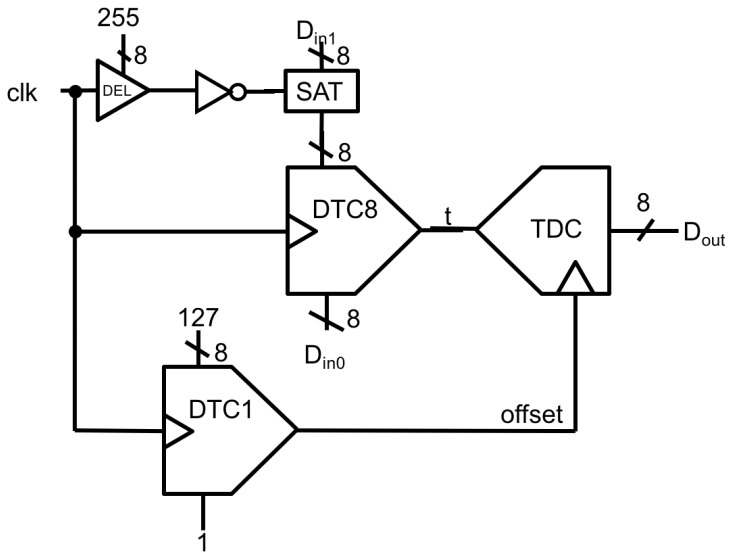


Figure 2.11: Multiplier.

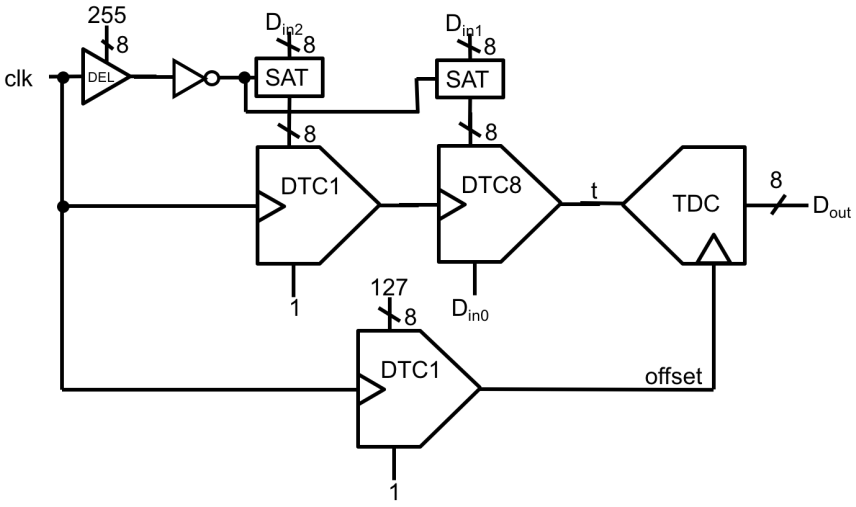


Figure 2.12: Multiply-add unit.

multiplication function. Once again, saturation logics are used to saturate the design.

2.4 Result

The MSMA is written in SPICE and simulated using HSPICE with FreePDK45 library. It is compared with a whole digital multiply-add unit (DMA), which is written in Verilog RTL and then converted to SPICE file using Design Compiler. The result of simulation is shown in this section.

2.4.1 Accuracy

In order to have high accuracy, first we have to make sure the delay is nearly linear to the data values. Figure 2.13 shows the delay of the 1-bit DTC. The R squared value of the fitted linear line is 0.99987, so we can conclude that the delay values are sufficiently linear.

To test the accuracy of the MSMA, we randomly sample 1000 different combinations of the input data which do not result in overflow in the multiply-add operation and calculate the error rate of the output data using the equation

$$\frac{|output - golden|}{255}$$

where *output* is the decoded output data from the MSMA and *golden* is the correct value. The result is shown in Figure 2.14 and the average error rate is **2.36%**.

2.4.2 Timing

Due to the fact that data value is represented in delay time in the time-domain, a long latency is inevitable. The main latency comes from TDC, where there is a long DTC and delay unit chain. To determine the minimum clock cycle, there are two constraints that should be met:

- The clock cycle, *cycle*, should be longer than the latency of the critical path. From Figure 2.7a and Figure 2.12, we can see that the inputs for the TDC, *in* and *clk*, are already delayed by a certain amount when entering the TDC. While inside the TDC, both signals go through a chain of the delay units and the DTCs. Therefore, the critical path under this constraint is from either *in* or *clk* to $D_{out}[0]$. To do the calculation, we need to add three parts together: The delay before the TDC, the delay of the delay units inside the TDC, and the delay of the DTCs inside the TDC. Since the delay depends on the multiply-add result, we first define $D_{in} = D_{in2} + D_{in1} \times D_{in0}$.

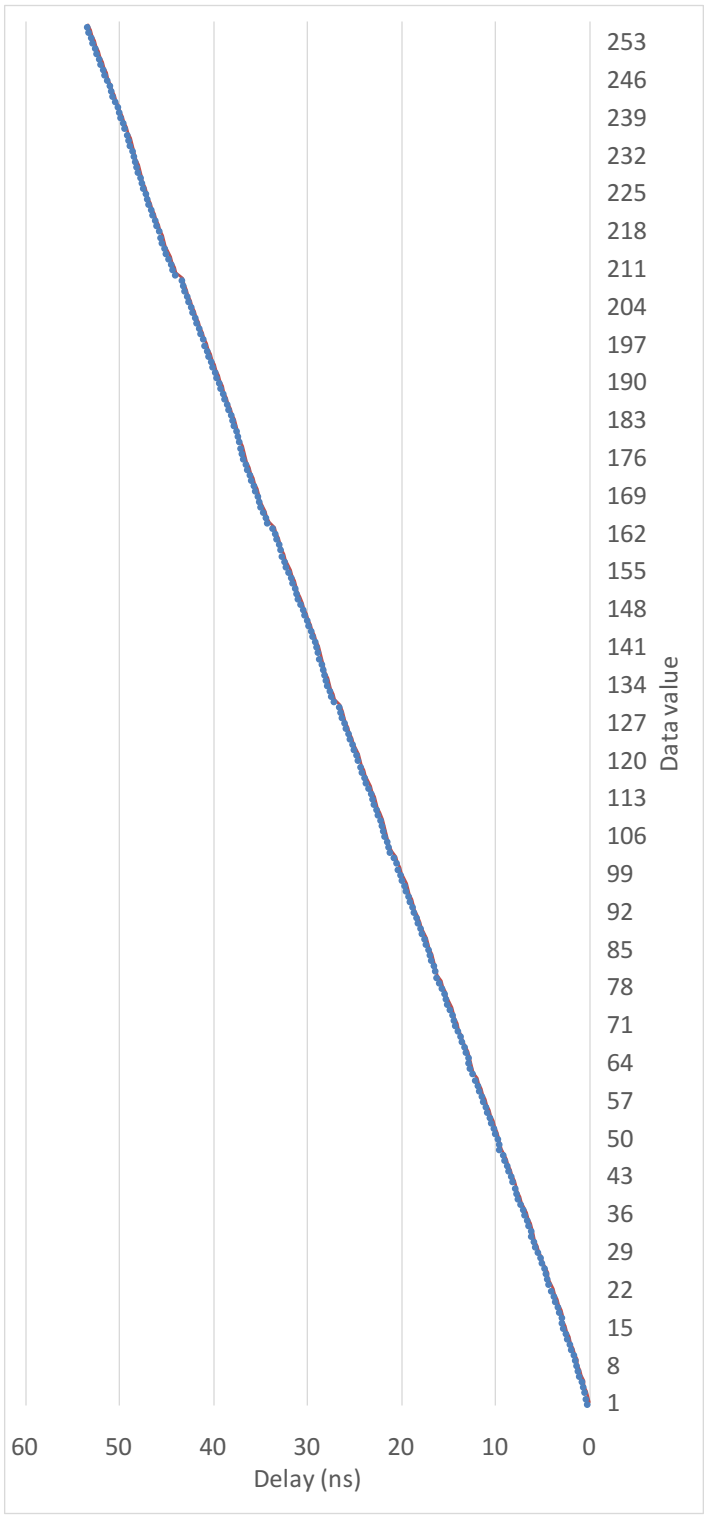


Figure 2.13: Delay versus data value in 1-bit DTC.

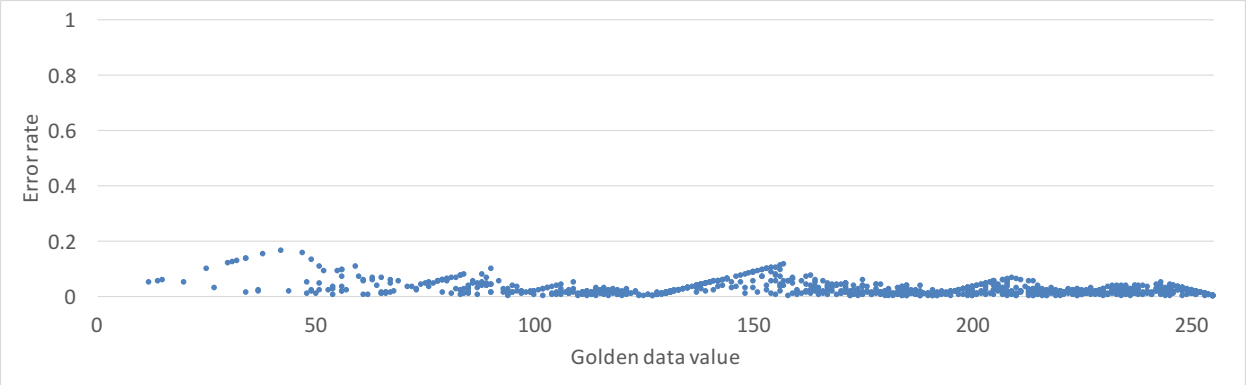


Figure 2.14: MSMA error rate of 1000 sample points.

Table 2.2: Clock cycle and frequency comparison of the MSMA and the DMA.

	MSMA	DMA
Clock cycle (ns)	250	1.4
Frequency (MHz)	4	714.3

- The delay before the TDC for in is $delay_t$, which depends on D_{in}; the delay before the TDC for clk is $delay_{offset}$.

$$delay_t = D_{in} \times T_{db}$$

$$delay_{offset} = 127 \times T_{db}$$

- The delay of the delay units are the same for both in path and clk path:

$$delay_{del} = 381 \times T_{db}$$

- The delay of the DTCs inside the TDC depends on D_{in}. Note that the delay for in path and the delay for clk path are complementary: Either one of them is delayed for each pair of DTCs.

The longest delay occurs when $D_{in} = 255$ and the critical path is the path that goes through in:

$$cycle > (255 \times T_{db}) + (381 \times T_{db}) + (0 \times T_{db}) = 636 \times T_{db} \tag{2.4}$$

- Due to the fact that the falling edge is not delayed by DTC (Figure 2.5), the clock signal has to remain high until the delayed signal going to the last DTC rises. As a result, the sum of the first and third part mentioned above has to be smaller than $\frac{cycle}{2}$. Since the largest sum is $255 \times T_{db}$, which is smaller than $381 \times T_{db}$, the condition is already met.

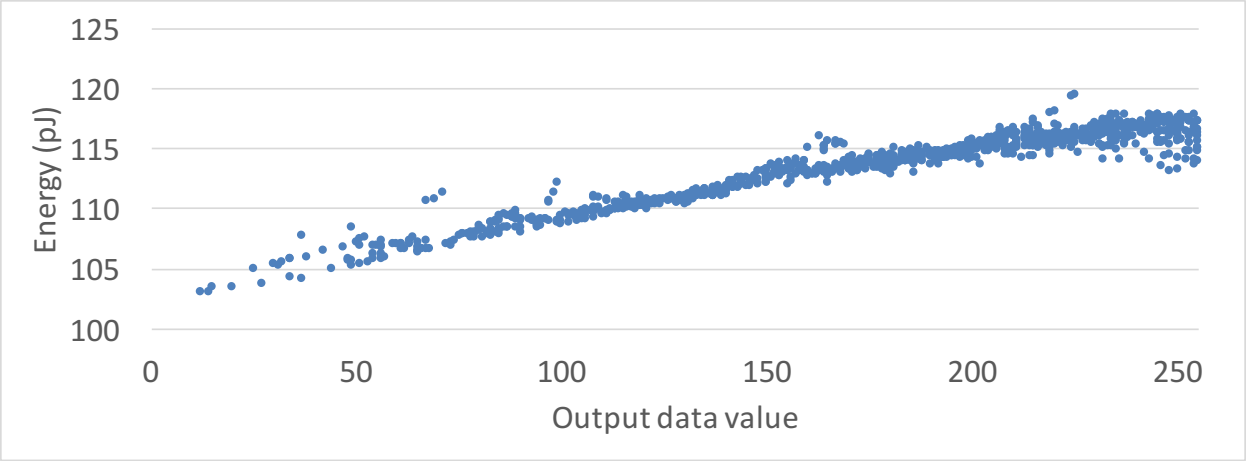


Figure 2.15: Energy of the multiply-add unit with respect to the output data value.

Table 2.3: Energy comparison of mixed-signal and digital units.

	Mixed-Signal	Digital
Energy (pJ)	113.24	0.73

Combining the two constraints, we get $cycle > 636 \times T_{db}$. As mentioned in Section 2.3.1, $T_{db} = 0.38$ ns, so the minimum clock cycle is approximately 242 ns. We use 250 ns eventually to reserve for possible delays in other gates and units. Table 2.2 shows the clock cycle comparison between the MSMA and the DMA. From the table it is obvious that the MSMA is about 180 times slower and therefore is one of the main reasons why time-domain analog and digital mixed-signal multiply-add is not as competitive.

2.4.3 Energy

Energy is another critical part of improvement of evaluation since we aim to design a more efficient multiply-add unit. Figure 2.15 shows the energy of the 1000 random samples, the same as the ones used for calculating the accuracy. Since one multiply-add operation is done per cycle, the energy is per operation and per cycle. Table 2.3 shows the average of the 1000 samples compared with the DMA. From the figure and the table, we can see that the energy of the MSMA consumes much higher energy, about 155 times more.

To reason about this, we remove the delay units of the MSMA because capacitors consume a lot of energy. From the simulation, the energy drops down to 1.01 pJ when the clock cycle is 250 ns. Therefore, we conclude that it is the capacitors that consume that much energy; however, because of the long clock cycle requirements, the energy consumption is still higher than the DMA.

2.5 Conclusion and Future Work

In this project, we designed and built a time-domain analog and digital mixed-signal multiply-add unit (MSMA) in the hope of replacing the merely digital one (DMA) with lower energy. We tested the functionality first in ModelSim using Verilog RTL, and then manually synthesized it to SPICE code and simulated in HSPICE. From the results of HSPICE simulation, we saw a few reasons why the MSMA is not better than the DMA:

- The energy is too high because of the capacitors in the delay unit.
- Even though the delay units are removed and replaced with other delay units that consume lower energy, the energy is still not lower than the DMA due to the long delay.
- If we try to shorten the delay (by shortening the base delay T_{db}), the accuracy will drop drastically because T_{db} will be in the same scale of the noise.

One possible way to deal with this is to shorten the delay by breaking an 8-bit MSMA into two 4-bit MSMAs and add the results in digital-domain. The delay decreases quadratically so the delay of the 4-bit operation is about 15 ns. In addition, pipelining can be added to the design. By applying these two techniques with pipeline depth 10, the speed is expected to be comparable with the DMA. In addition, with the delay units realized by transistors rather than capacitors, the energy can potentially be much lower than the digital unit because of the short delay and pipelining. Last but not least, the accuracy can be better due to the fact that part of the computation is moved back to digital-domain.

To summarize, the MSMA we propose does not successfully enjoy the benefits from both time-domain and digital-domain. In order to gain advantages from both domains, the balance is critical and the distribution of the operations in the two domains needs to be taken good care of. The proposed future work may have better balance, but it will not be known until the work has been done.

Chapter 3

Cyclic Redundancy Check Hash Table Lookup

3.1 Overview

Hash tables have been widely used to serve as caches to hold temporary data that will very likely be used again in the future. When the size is small enough, the short search time of the hash table is well appreciated as well as its low energy. Therefore, we manage to take those advantages and replace the multiply-add computations with hash table lookups. We choose cyclic redundancy check (CRC) as our hash function. CRC is an error-correcting code used commonly in digital devices to detect the occurrence of unexpected data change. However, it is not only used in error-correcting but also in hashing because of its simplicity of implementation in hardware and its great ability to distribute the data. In this project, our simulation shows that the CRC hash table reaches 1.96 energy improvement with accuracy drop no more than 1.2%.

3.2 Introduction

CRC is the remainder of the polynomial division of the original data being attached to and, upon retrieval, the destination device does the same remainder calculation and compares the codes to detect the errors. The name of the CRC comes from the fact that the code is calculated based on **circular** shift and the code is **redundant** in terms of the information given in the data. CRCs are popular because the computation is very simple to implement in hardware.

CRCs are also used for hashing because of the following benefits:

- **Simple to implement in hardware**: As mentioned before, the computation of CRC is simple because it contains nothing more than shifting, exclusive-oring and 256-entry table accessing. Moreover, in some cases table accesses can be removed. This extremely simple hardware implementation strongly lowers the execution time and power consumption.
- **Able to handle any sequence length**: Unlike most of the hash functions which require fixed-length input sequences, CRC can handle an arbitrary input length. This flexibility removes extra hardware

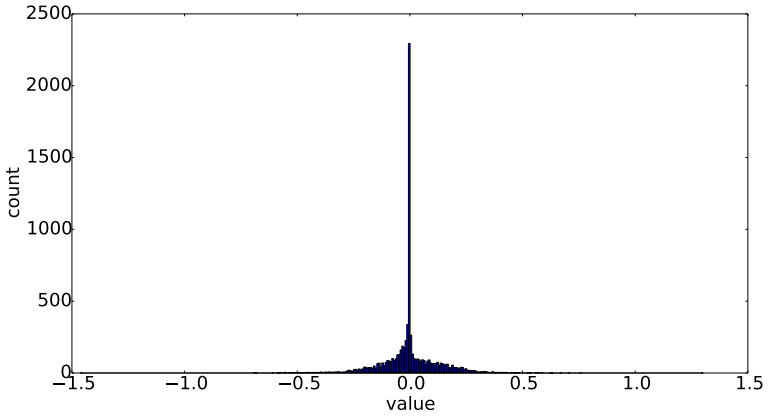


Figure 3.1: Weight distribution histogram of the NN trained for MNIST.

Table 3.1: Rates of the magnitude of weights smaller than the thresholds.

Threshold	MNIST	Hotspot
0.2	88.6%	100%
0.5	99.5%	100%

or software to generalize the hash function to consume sequences with arbitrary length.

- **Low collision**: Collision is one of the key points to consider when choosing the hash function. A good hash function with low collision rate is very important to avoid useful data being evicted. A good hash function distributes the data evenly and randomly, and from the previous study [Jain, 1992], CRC is a great fit for such requirements.

As explained in Chapter 1, DNN has few dominant operations, and the leading one is matrix-multiplication, which is composed of multiply-add operations. Our goal is to improve the DNN architectures by optimizing this major computation. The existing units that do the operation, namely the arithmetic logic unit (ALU) and the floating point unit (FPU), consume lots of energy, and it often takes a few cycles for the FPU to do multiplication. As a consequence, we aim to eliminate the computations by replacing them with table lookups. The reasons for choosing lookup tables as replacements are as follows:

- Although there are 2^{32} possible different values in one input data, most of the values are very rarely seen, especially when the data represents the value of the weights and neuron values in NNs. For example, Figure 3.1 shows the weight distribution of the NN trained for the MNIST benchmark.[1] From the figure, we can see that most of the weights are close to zero. Table 3.1 shows the rate of the weights being within the range of 0.2 and 0.5 of benchmarks MNIST and Hotspot.[2] In floating

[1] http://yann.lecun.com/exdb/mnist/
[2] http://www.cs.virginia.edu/skadron/wiki/rodinia/index.php/HotSpot

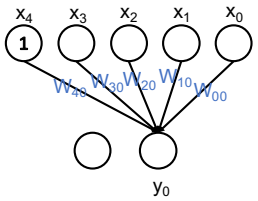


Figure 3.2: An example of a small feed-forward neural network. x_i refers to the input neuron, and W_{ij} refers to the weight.

point representation, there are only about 250 data sequences used to represent the value with the magnitude smaller than 0.5. Therefore, due to this highly clustered feature, there is a guarantee of high hit rate in the table lookup.

- In the previous project, the multiplication is limited to integer data type because of the analog feature. This introduces overhead of transforming the data type from floating point to integer. In addition, the accuracy drops because of the data type change. We aim to gain energy improvements with lossless accuracy, and the result is the hash table lookup which can handle floating point data type.

The rest of the chapter is organized as follows: Section 3.3 describes the CRC hashing method, the hash table structure, and how we incorporate them into NN computations; Section 3.4 shows the simulation results of our implementation; finally, Section 3.5 reviews the design and draws the conclusion from the results.

3.3 Implementation

Figure 3.2 shows a small example of a simple fully-connected NN: the output neuron y_j equals $\sum_{i=0}^{n-1} x_i \times W_{ij}$, where i is the index of the input neuron, j is the index of the output neuron, and n is the number of input neurons. The last weight is in fact the bias, and the value of the corresponding "fake" input neuron is one. If we take the example shown and expand the equation, we get

$$y_j = x_0 \times W_{0j} + x_1 \times W_{1j} + x_2 \times W_{2j} + x_3 \times W_{3j} + x_4 \times W_{4j}$$

The lookup table replaces the $\times$ and $+$ operations. Specifically, if the number of inputs is two, then the equation becomes

$$y_j = lookup(x_0, W_{0j}) + lookup(x_1, W_{1j}) + lookup(x_2, W_{2j}) + lookup(x_3, W_{3j}) + lookup(x_4, W_{4j})$$

If the number of inputs is four, then

$$y_j = lookup(x_0, W_{0j}, x_1, W_{1j}) + lookup(x_2, W_{2j}, x_3, W_{3j}) + lookup(x_4, W_{4j}, 0, 0)$$

and so on.

In this section, we discuss our design: the calculation of the hash key, and the architecture of the hash table. In order to achieve the highest hit rate, we choose CRC as our hash function and tune the following parameters:

- **Number of inputs**: Number of inputs being hashed. More inputs require fewer table lookups, but result in lower hit rate or lower accuracy.
- **Hash table size**: The size of the hash table. The hit rate will be too low if the size is small; the energy consumed will explode if the size is too big.
- **CRC length**: The length of CRC hash key, which depends on the hash table size.
- **Tag length**: The length of the tag used for tag comparison in the hash table. The shorter the tag length, the higher the hit rate and the lower the accuracy.

In the remainder of the chapter we will elaborate the meaning of these parameters, discuss the methodology to tune the parameters, and present the final values of the parameters.

3.3.1 Hash Scheme

CRC hashing works by calculating the remainder of the input sequence divided by a polynomial. The polynomial is carefully selected so as to map the input sequence evenly and randomly onto the hash key space. An n-bit CRC means the hash key, namely the remainder, is n-bit long, and is often denoted as CRC-n. Since the polynomial is purposely picked, the calculation can be simplified based on the polynomial. For instance, CRC-16 hash function with polynomial $x^{16} + x^{12} + x^5 + 1$ can be simplified to many identical updates with each update implemented as:

```
s = d ^ (c >> 8);
t = s ^ (s >> 4);
r = (c << 8) ^ t ^ (t << 5) ^ (t << 12);
```

where c is the hash key before one update, r is the hash key after that update, and d is the incoming byte data. The input sequence is broken down to an array of byte data. One byte data is being fetched upon each update and the hash key is being updated until all the byte data are used.

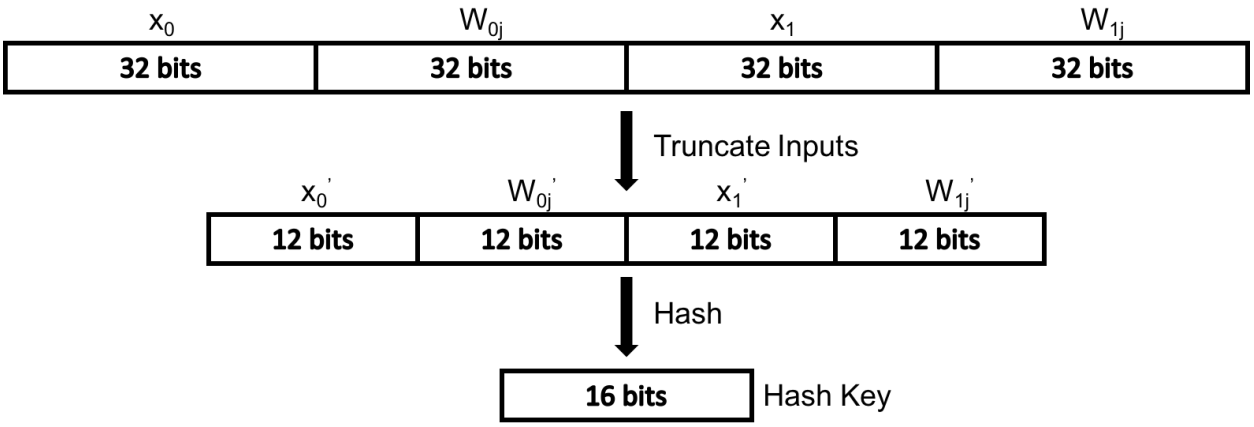


Figure 3.3: An example of hashing: 4 inputs.

From the algorithm description above, it is obvious that the input sequence, as well as the hash keys, can have arbitrary lengths. However, if the input sequence is much longer than the hash key, the chance of collision becomes higher.

The steps for acquiring hash keys are described as follows and illustrated in Figure 3.3:

- **Truncate inputs**: As mentioned in Chapter 1, DNNs can tolerate inaccurate data values. Therefore, to reduce the collision introduced by a long input sequence, we remove the bits that give less information from the input sequence, which is the LSBs of the mantissa.
- **Hash**: Hash the truncated input sequence with the algorithm described above into a 16-bit hash key.

3.3.2 Lookup Table

Figure 3.4 shows the structure of multiply-add operation with the hash table. The input sequence is mapped to a hash key. The hash key is then split into the index part and the tag part. The number of bits for the index, x, is determined by the number of entries in the hash table. The rest of the bits are then considered as tag bits. The output data will be provided by the hash table if there is a hit; otherwise, the FPU is enabled to provide the data and the result will be written into the table.

The hash table used here is different from the traditional hash table: traditional hash tables resolve the collision by either employing a linked list or probing. This hash table does not resolve the collision, however, since inaccurate values are acceptable and therefore the input sequences that are mapped to the same hash key are treated as exactly the same value. This non-one-to-one mapping introduces some errors to the result. However, if the parameters (number of inputs, hash key length, and hash table size) are chosen properly, the error is negligible. The benefits achieved from this change are:

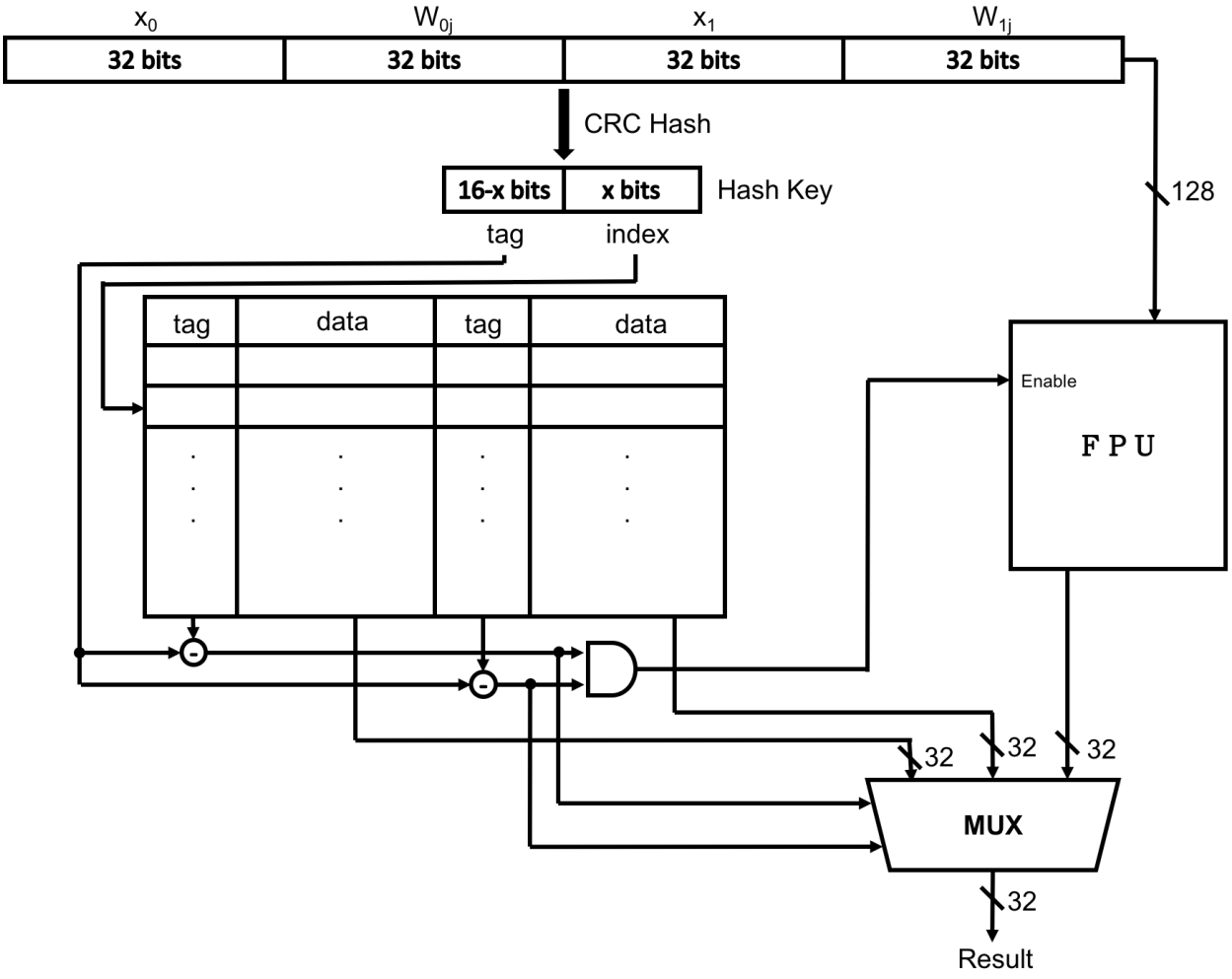


Figure 3.4: Block diagram of multiply-add operation with a two-way associative hash table.

Table 3.2: Benchmarks used for simulation.

Benchmarks	MNIST	Hotspot
Description	Recognizing images of handwritten digits	Computing CPU temperature
Input of NN	28×28 image	3×3 temperature grid
Output of NN	List of probability of the matching digit with size 10	Temperature value
Accuracy of trained NN	90.19%	99.38%

Table 3.3: Tools used to estimate timing and energy.

Subject	Hash table	Hash functions	Floating point multiply-add
Tool	OpenRAM [Guthaus et al., 2016] + Design Compiler	Design Compiler	Design Compiler

- **The hash table is well-structured**: Unlike the traditional hash table structured as a linked-list, this hash table is cache-like, where all the data are stored together and benefit from temporal and spatial locality.

- **Lookup takes O(1) time**: The two traditional methods of resolving collision both have the main disadvantage of long lookup time. In linked-list, the data with the same bucket number are chained in a long list, so looking up takes more time if the list is longer. In probing, the data migrates to another bucket if there is a collision, making the searching more difficult. Our hash table does not have a long list in a bucket; nor does it move data. The lookup only takes one shot.

- **Small tags save storage**: Due to the fact that all the input sequences that are mapped to the same hash key are treated the same, the hash key can be used as a unique ID number. Therefore, the tag that originally was the input sequence is now replaced with the tag portion of the hash key (the index portion is used as the address of the hash table), which reduces the tag length from $n \times 32$ to $16 - x$, where n is the number of inputs. Moreover, if the hash table is direct-mapped, the tag part disappears. This saves a lot of space in the hash table especially when the input sequence is long.

3.4 Results

We write a functional and performance simulator in Python to simulate the NNs with hash tables and compare with the NNs using merely FPU. Table 3.2 lists the benchmarks we use and Table 3.3 shows the tools we use to estimate energy. Note that we use a trained NN for each benchmark as our baseline and so the accuracy of the trained NN is not 100%. Theoretically, the accuracy with hash tables cannot be higher than the baseline accuracy. However, in the following sections we sometimes see the accuracy being higher than the baseline accuracy due to the fact that the value read from the hash table can be different from the value from the FPU but happen to be more accurate.

Table 3.4: Energy estimation from OpenRAM for different hash table size.

Size (kB)	1	2	4	8	16
Energy (pJ)	1.947	1.955	1.971	2.369	2.797

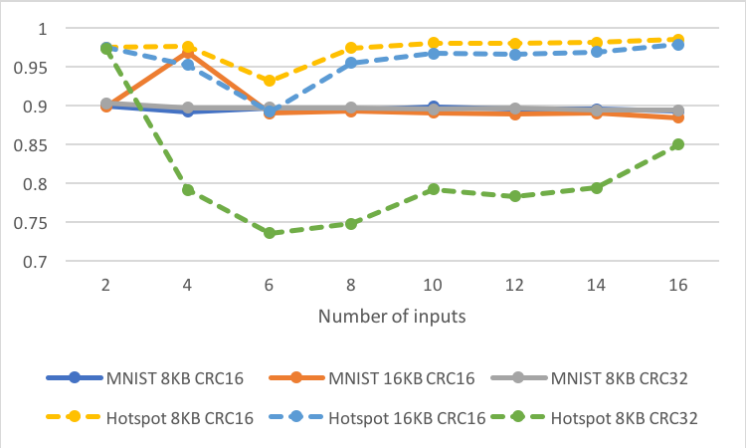
The following sections show the result of tuning each parameter and the final optimal value.

3.4.1 Number of Inputs

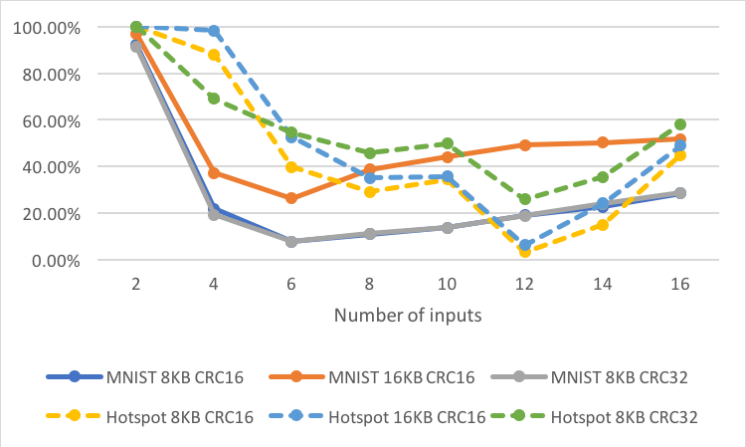
Figure 3.5 shows the results of sweeping the number of inputs from 2 to 16 (replacing the number of operations from 1 to 8). The hit rate result is as expected: when the number of inputs is two, the hit rate is high because the chance of multiplying two same numbers is high; as the input sequence gets longer, the hit rate drops drastically because the chance of having two same multiply-adds is much smaller than having single same multiply-add. However, the hit rate starts to rise slowly after six inputs. This is because of the collision hit in the hash key: the chance of two different sequences colliding to the same hash key gets high, and in our design, the input sequences are considered exactly the same if the corresponding hash key is the same, thus resulting in a hit in the hash table lookup. This helps save energy, which is why the energy improvement also increases slightly as the number of inputs gets larger; however, it may cause a little accuracy drop. Nevertheless, regardless of the trend as the number of inputs grows, the energy improvement of two inputs is overwhelmingly better than the energy improvement of more inputs and the accuracy of two inputs is lossless. Hence, we choose two as the number of inputs.

3.4.2 Hash Table Size

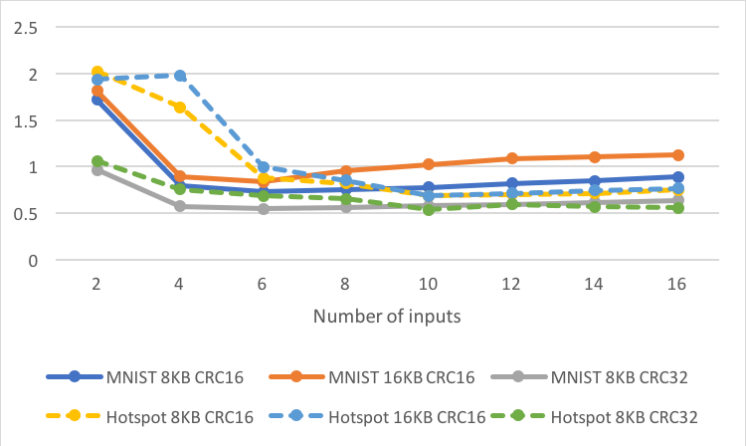
Table 3.4 shows the estimated energy of different hash table size. We run the simulation from 2 kB to 16 kB and the results are in Figure 3.6. From the figure, the trends for MNIST and Hotspot are totally different: Hotspot has a rather small input size, output size, and small NN. Hence, from the fact that the hit rate saturates after 4 kB, we can tell the Hotspot NN only requires a small size of memory. On the other hand, the hit rate of MNIST keeps increasing, indicating that the memory footprint is larger than 16 kB. We then conclude that the best hash table size is application dependent. However, as the data grows and new complex algorithms emerge, the NN size is often big; therefore, it is often better to choose 16 kB over 4 kB.



(a) Accuracy

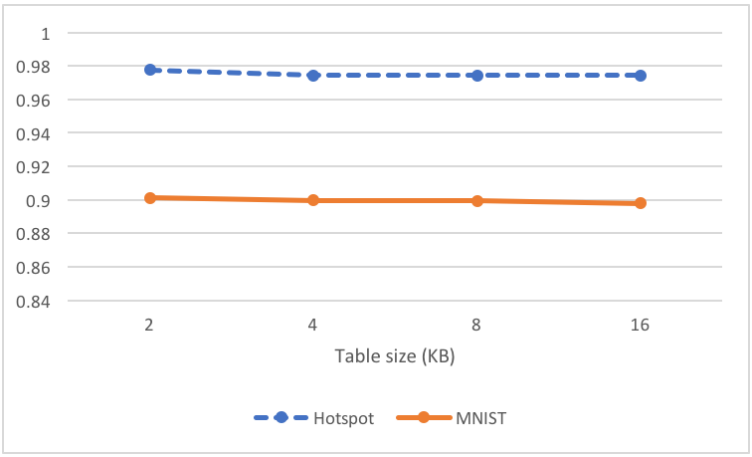


(b) Hit rate

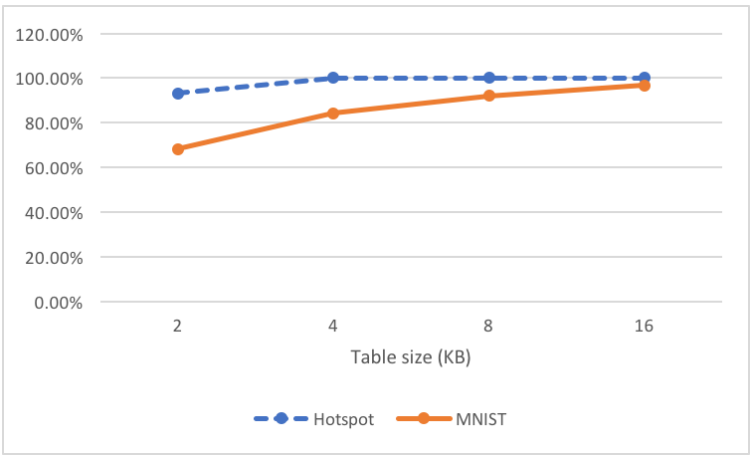


(c) Energy improvement

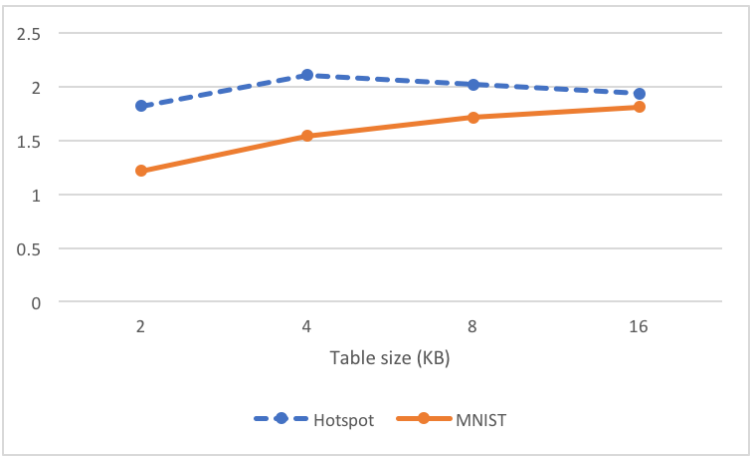
Figure 3.5: Results of sweeping the number of inputs from 2 to 16.



(a) Accuracy



(b) Hit rate



(c) Energy improvement

Figure 3.6: Results of sweeping the table from 2 kB to 16 kB. The number of inputs is two and the CRC length is 16 bits.

Table 3.5: Results of hash tables with different tag length.

Benchmark	MNIST(two-input, 16KB, CRC-16)			Hotspot(two-input, 4KB, CRC-16)		
	Accuracy	Hit rate	Energy improvement	Accuracy	Hit rate	Energy improvement
Original	89.94%	92.08%	1.72	97.43%	99.96%	2.11
Reduced	89.81%	92.17%	1.72	85.58%	99.96%	2.11

3.4.3 CRC Length

Part of the CRC hash key is used as an address to index the hash table; hence the CRC hash key must be at least as long as index bits. In addition, the tag part of the key is used to search for the matching input sequence in the entry that index part points to; thus, to avoid conflict misses, the length of the tag bits should be as close as $log_2(associativity)$. Since the hash tables we are aiming at here are small (no larger than 16 kB), the hash key is calculated to be best when it is 16 bits long.[3] Plus, the energy of generating CRC-32 hash key is larger than that of CRC-16 because of more total gates required in CRC-32.

To verify our assumption, we also simulate with CRC-32 and the results are also shown in Figure 3.5. Comparing CRC-16 and CRC-32 in MNIST (blue and gray solid lines), the accuracy and hit rate are almost the same, meaning that CRC-32 distributes the data as well as CRC-16. However, the energy improvement of CRC-16 is much better than that of CRC-32. In Hotspot (yellow and green dashed lines), CRC-32 distributes data poorly and results in high hit rate and poor accuracy, indicating the collision rate is high. In addition, the energy improvement is worse than CRC-16. All the results match our expectation and therefore CRC-16 is a better choice over CRC-32.

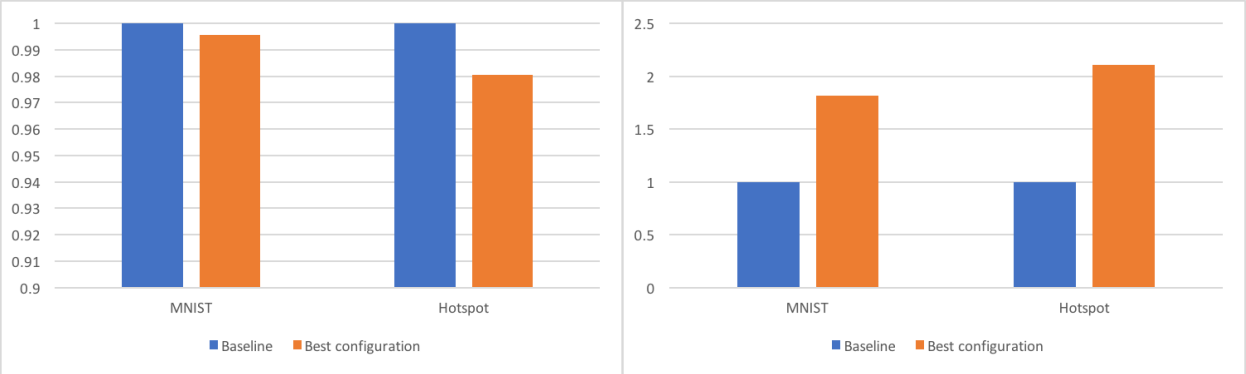
3.4.4 Tag Length

Tag length determines the accuracy and hit-rate trade-off. The original tag length is given by CRC length minus number of index bits. For example, for a CRC-16 hash table with 2^8 entries, the tag length is $16-8=8$. However, if the hit rate is too low where there is no energy improvement, the tag can be shortened so that more different input sequences are mapped to the same tag and thus the hit rate is increased. Nevertheless, the increase of hit rate comes from the increase of collision hits, which causes accuracy drop. In the results we obtained, the energy improvement is already good enough that little shortening is needed. Despite this fact, we still run the simulation where the tag length is reduced by two and the result is shown in Table 3.5. We choose the best parameters for the two benchmarks and compare the results of different tag lengths. The result does not change much in MNIST, but the accuracy drops massively in Hotspot.

[3] A 16 kB hash table has 2^9 entries and is 4-way associative, so the key must be at least 11 bits long and the closest is CRC-16.

Table 3.6: Best hash table configuration for the benchmarks.

Parameters	Number of inputs	Table size	CRC length	Tag length
MNIST	2	16 kB	16	7
Hotspot	2	4 kB	16	9



(a) Normalized Accuracy

(b) Normalized energy improvement

Figure 3.7: Normalized accuracy and energy improvement of best configurations.

3.4.5 Summary

The normalized accuracy and energy chart chosen from the best parameters (Table 3.6) is shown in Figure 3.7. On average, the hash table can reach up to **1.96** energy improvement with only **1.2%** accuracy lost.

3.5 Conclusion and Future Work

In this project, we improve the NN architecture by substituting the FPU, which does multiply-add operation, with hash table lookup, to substantially lower the energy consumption on average by 1.96× while at the same time limiting the accuracy drop to **1.2%**. With this promising result, the NN accelerator can be further optimized by replacing complicated activation functions that require non-linear computation including divisions, with hash table lookups. Some of the activation functions, such as ReLU function, converge to a certain value as the input value goes to infinity. This feature is especially preferable for hash table in that it requires little memory. With proper data truncation and mapping, expanding this project to eliminating activation function computation is expected to achieve significant energy improvement with very little accuracy loss.

Chapter 4

Conclusion

In this thesis, we propose two different methods to replace the traditional digital multiply-add operations done in either ALU or FPU. The first method is to build a time-domain analog and digital mixed-signal multiply-add unit because the analog circuit is considered to consume much less power than the digital circuit, and we try to leverage that with the high performance of the digital circuit. However, the result is not as promising because the analog circuit is way too slow and therefore the energy consumption does not go down as intended. Possible improvements can be done to integrate time-domain and digital-domain better in order to beat the traditional digital multiply-add operation. The second method utilizes the hash table to eliminate power-hungry multiply-add computations. Rather than actually doing the computations, this method instead looks for the records of the previous computation values in the hash table and uses the value of the matching record. The simulation shows that this method consumes 1.96× less energy on average while maintaining the accuracy loss within 1.2%. With this satisfactory result, further optimization can replace the compute-intensive activation functions with the same technique.

References

[Esmaeilzadeh et al., 2011] Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., and Burger, D. (2011). Dark silicon and the end of multicore scaling. In *Proceedings of the 38th International Symposium on Computer Architecture*.

[Esmaeilzadeh et al., 2012] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. (2012). Neural acceleration for general-purpose approximate programs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*.

[Guthaus et al., 2016] Guthaus, M. R., Stine, J. E., Ataei, S., Chen, B., Wu, B., and Sarwar, M. (2016). Openram: An open-source memory compiler. In *Proceedings of the 35th International Conference on Computer-Aided Design (ICCAD)*.

[Hinton et al., 2012] Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., and Kingsbury, B. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29:82–97.

[Jain, 1992] Jain, R. (1992). A comparison of hashing schemes for address lookup in computer networks. *IEEE Transactions on Communication*, 40:1570–1573.

[Jouppi et al., 2017] Jouppi, N. P., Young, C., and Patil, N. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*.

[Minsky and Papert, 1969] Minsky, M. and Papert, S. A. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press.

[Miyashita et al., 2014] Miyashita, D., Yamaki, R., Hashiyoshi, K., Kobayashi, H., Kousai, S., Oowaki, Y., and Unekawa, Y. (2014). An ldpc decoder with time-domain analog and digital mixed-signal processing. *IEEE Journal of Solid-State Circuits*, 49(1):73–83.

[Reagen et al., 2017] Reagen, B., Adolf, R., Whatmough, P., Wei, G.-Y., and Brooks, D. (2017). *Deep Learning for Computer Architects*. Morgan & Claypool.

[Sampson et al., 2011] Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., and Grossman, D. (2011). EnerJ: Approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*.